REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
08-03-2004	Final Report	30 August 2002 - 29-Feb-04

4. TITLE AND SUBTITLE

Agents Inaccessibility in Multi-Agent Systems

5a. CONTRACT NUMBER
FA8655-02-M4057

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Professor Vladimir Marik

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Czech Technical University of Prague
CTU FEL
Technická 2,
Praha 6 166 27
Czech Republic

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
SPC 02-4057

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

This report results from a contract tasking Czech Technical University of Prague as follows: The contractor will investigate the combination of two novel approaches to address the inaccessible agent problem; (i) social knowledge and acquaintance models and (ii) a doubler agent concept. As detailed in the technical proposal, the work will entail four primary tasks: (1) domain analysis, (2) short term inaccessibility, (3) long term inaccessibility, and (4) testing and validation.

15. SUBJECT TERMS
EOARD, Multi Agent Systems, Behavior Based Prediction, Agent Based Systems

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18, NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			PAUL LOSIEWICZ, Ph. D.
UNCLAS	UNCLAS	UNCLAS	UL		19b. TELEPHONE NUMBER *(Include area code)* +44 20 7514 4474

Inaccessibility in Multi-Agent Systems

contract number: FA8655-02-M-4057

principal investigators:

Michal Pěchouček and Vladimír Mařík

team members:

David Šišlák, Martin Rehák, Jiří Lažanský and Jan Tožička

Gerstner Laboratory,
Czech Technical University in Prague

http://gerstner.felk.cvut.cz

Inaccessibility in Multi-Agent Systems

Deliverable d.3 – Final Report

Michal Pěchouček, David Šišlák, Vladimír Mařík, Martin Rehák, Jiří Lažanský and Jan Tožička

Gerstner Laboratory, Agent Technology Group
Department of Cybernetics, Czech Technical University in Prague,
Technická 2, 166 27, Prague 6, Czech Republic

Abstract. The "Inaccessibility in Multi-Agent Systems" project (contract no.: FA8655-02-M-4057) has been investigating the problem of agents' temporally or permanent inaccessibility. Within the project the problem of agents inaccessibility has been studied, the concept of agent's acquaintance models and their use in the partially inaccessible environment has been investigated, the model of the stand-in agent has been designed and implemented and primarily the $\mathcal{A}$–GLOBE agent interaction platform (formerly denoted as AIT - Agent Inaccessibility Testbed) has been developed. This is the **deliverable** d.3, that provides technical information about the pillar achievements of the project.

Table of Contents

1 Introduction 4
1.1 Project Plan and Statement of the Technical Work 4
2 Inaccessibility 6
2.1 Classification of Inaccessibility 6
2.2 Measuring Inaccessibility 8
3 Sharing Social Knowledge 12
3.1 Acquaintance Model 12
3.2 Stand-in agent 13
4 Meta-Reasoning 16
4.1 MRinMAS Project 16
4.2 Integration with the Project 17
5 Experiments 18
5.1 Description of the experimental settings 18
5.2 Experimental results 21
6 $\mathcal{A}$–GLOBE Agent Integration Platform 26
6.1 Upgrade from AIT to $\mathcal{A}$–GLOBE 26
6.2 Functional Description 27
6.3 System Architecture 27
6.4 Agent Platform 28
6.5 Agent container 30
6.6 Services 41
6.7 Environment Simulator 42
6.8 Sniffer Agent 47
7 Platform comparison 48
7.1 Agent platforms characteristics 48
7.2 Message speed benchmarks 49
7.3 Memory requirements benchmark 50
7.4 $\mathcal{A}$–GLOBE Summary Statement 52
8 Conclusion 53
9 Declaration 53

1. Introduction

The "Inaccessibility in Multi-Agent Systems" project (contract no.: FA8655-02-M-4057) has been investigating the problem of agents' temporally or permanent inaccessibility. Within the project the problem of agents inaccessibility has been studied, the concept of agent's acquaintance models and their use in the partially inaccessible environment has been investigated, the model of the stand-in agent has been designed and implemented and primarily the $\mathcal{A}$–GLOBE agent interaction platform (formerly denoted as AIT - Agent Inaccessibility Testbed) has been developed. This is the **deliverable** d.3, that provides technical information about the pillar achievements of the project.

1.1. Project Plan and Statement of the Technical Work

The project has been structured into four mutually interlinked work-packages (cited from the project proposal):

1. **Domain analysis** (month 1 - month 4): Firstly, the problem of agents inaccessibility in multi-agent systems will be thoroughly analyzed and investigated. This analysis will be carried out on a general level, but also with respect to the domain of planning of humanitarian relief operations, the functionality of the CPlanT multi-agent system being considered. The Sufferterra humanitarian relief scenario will be redesigned accordingly (in the extent needed).

2. **Short term inaccessibility** (month 5 - month 11): Potentials of the classical technology of social knowledge and acquaintance models will be studied for the agent's short term inaccessibility. Robust and reliable algorithms for identifying short-term malfunctions of agents as well as their unavailability and techniques for reconstructing the status of the unavailable agent will be designed. Agent's short term inaccessibility will be studied from the inaccessible agent's point of view and also from the viewpoint of the rest of the community.

3. **Long term inaccessibility** (month 8 - month 15): Limitations of the technology developed in the WP2 for agent's long term inaccessibility will be studied. The core of this WP will be the design of the formal model of the doubler agent. Applicability of meta-reasoning, that is a subject of the AFOSR research project running in parallel - "Monitoring and Meta-reasoning in the Multi-Agent Systems", for implementing advanced reasoning capabilities of the doubler agent will be investigated. The doubler agent will be implemented in the form of a research prototype.

4. **Testing and validation** (month 16 - month 18): Capabilities of the developed and implemented technologies will be verified for agents' short-term and long-term inaccessibility in the CPlanT multi-agent system. Testing and validation will be carried out in a quantifiable way. Appropriate measures and criteria that will be designed and proposed within this WP.

The project has achieved the set research goals and several additional (non-planned) contributions have been achieved during the project. As it is very often so in research project, we had to face several unexpected occurrences that caused small deviation from the plan.

It has been shown that the CPlanT multi-agent system does not support mobility and handling possible agent's inaccessibility. For that reason it has not been possible to re-use the CPlanT multi-agent infrastructure for experiments and for validation of the investigated concepts.

We have also failed to select a commercially available infrastructure (there is none available) as a testbed for the project works. Consequently, we have decided to design and develop an Agent Inaccessibility Testbed (AIT). As the testbed is carefully designed to be fully open for experimentation with different techniques and problems of distributed computing and multi-agent systems, we have transformed the testbed into a fully functional agent interaction platform: $\mathcal{A}$–GLOBE. This platform is an unexpected by-product of the project.

It has turned out that studying the concepts of short-term and long-term inaccessibility jointly is advantageous. Therefore the phases 2 and 3 has been integrated within the project. The project team working on this project has very closely collaborated with the researchers that worked on the research contract Monitoring and Meta-Reasoning in Multi-Agent Systems (MRinMAS), contract number: FA8655-02-M4056. The transfer of research results has taken place bidirectionally. The meta-reasoning techniques have been investigated within this project for solving agents' inaccessibility and the $\mathcal{A}$–GLOBE platform is used within the MRinMAS project for testing and experimenting with meta-reasoning and monitoring.

According to the plan the set of deliverables have been shipped to AFRL:

- **month 3** – domain analysis, formal model if inaccessibility,
- **month 15** – research prototype, system documentation
- **month 18** – this report, the final deliverable[1].

[1] This report is a final deliverable and contains a unified summary of the technical results achieved within the project. Therefore, some parts (very few) of the report may have been already included in the previous deliverables.

2. Inaccessibility

Agents' inaccessibility in a multi-agent community is an uneasy problem of a high practical importance. There are several different reasons why an agent may become inaccessible by the other members of the multi-agent community - such as malfunction of the communication links, communication traffic overload, agent leaving the communication infrastructure for accomplishing a specific mission, agent failing to operate, etc. Agents do act autonomously, therefore we can speculate and predict, we can hardly prove agents future behavioral patterns (which is given by their autonomy). Similarly we cannot guarantee complete reliability of the communication links and the communication infrastructure in highly distributed systems. There are also agents who work 'off-line' and still wish to be acknowledged as fully fledged community members. Consequently, there is a need for a unified and general technology for maintaining social stability/sustainability in multi-agent system with inaccessible agents.

Within the frame of this project we have been investigating the combination of two different approaches of distributed artificial intelligence - (i) the concept of **social knowledge** and **acquaintance models** and (ii) the concept of various **middle agents** (e.g. doubler agent, stand-in agent, matchmakers, mediators etc.) for solving this problem.

2.1. Classification of Inaccessibility

Systematically we distinguish between several classes of types of inaccessibility. Inaccessibility can be caused by:

1. unreliability of the communication infrastructure: periodical, singular or permanent communication lags and drop-outs,
2. balancing the cost of the communication: when an objective function is associated with sending messages (e.g. cost, time, energy resources, etc.),
3. dynamic changes of the communication infrastructure: when platforms or agents physically change their location and thus availability with respect to different communication means,
4. partial communication infrastructure: when all nodes are not fully interconnected,
5. semi-collaborative environment: when agents agreed to communicate only a certain type of information, while other keep private (here we talk about inaccessibility of information rather than agent),

6. fully adversarial environment: when agents simply do not want to communicate even if accessible.

These types of agents' inaccessibility can occur in different possible situations:

- **agent physical mobility:** Agents may physically leave the community. This can happen in the situations where intelligent agents resides on mobile platforms such as cell phones, PDA or any other IP-addressable devices carried e.g. by a soldier, or mobile vehicles. For inaccessibility to happen, the communication infrastructure needs to be incomplete due to hilly area, large distances, etc. This scenario is based in inaccessibility type 3 and 4.

- **agent code migration:** Strong and weak migration of the software code initiate inaccessibility only within communication infrastructures that are not completely reliable. Software agents may need to migrate e.g. in the situations where the information needs to be transformed between two mutually inaccessible locations connected by route that use mobile vehicles. Such migration may lengthen a communication link between agents, increasing its potential to fail in the future.

- **ad-hoc networks:** Specific situations may happen when intelligent agents residing on either mobile vehicle or soldiers' equipment need to communicate but they have only short-range broadcasting equipment. The movement of agent carriers may be dependent or completely independent from the agents communication requirements. Situations may happen when the agents' carriers need to physically arrange their location so that an information 'pontoon bridge' for a communication relay can be facilitated.

- **adversarial environment:** In a hostile environment the situations may occur when the agents refuse to communicate in order to keep their existence undisclosed. Such an agent is then regarded as inaccessible. There is a variation of this scenario, when agent can receive the information while cannot send any information, e.g. when broadcasting a signal may disclose agents physical location, while receiving is a passive activity that does not reveal agent's position.

- **unreliable communication networks:** Often there are requirements for communication in the networks where communication is imperfect. This is the case of complex utility networks (administered by different owners), supply chains, or information infras-

tructures in the underdeveloped regions. Inaccessibility is here of a types 1 and 2.

– **information inaccessibility:** A specific situation is when agents agree to communicate only specific type of information (semi-private see [12]) and keep the private information confidential. Private information is regarded as inaccessibility. This scenario is typical for the information fusion and intention modelling problems.

2.2. Measuring Inaccessibility

An important problem is how to quantify inaccessibility in multi-agent systems. In the following we discuss several metrics of inaccessibility that we have been using throughout our research project.

Let us introduce a **measure of inaccessibility**, a quantity denoted as $\overline{\vartheta} \in < 0; 1 >$. This measure is supposed to be dual to the **measure of accessibility** – $\vartheta \in < 0; 1 >$, where $\vartheta + \overline{\vartheta} = 1$. We will want ϑ to be 1 in order to denote complete accessibility and ϑ to be 0 in order to denote complete inaccessibility.

First, we may define this concept on a single pair of directly connected agents by using two different underlying formalisms. First, **direct accessibility** may be defined using the uptime of the link connecting these two agents:

$$\vartheta_t = \frac{t_{\text{acc}}}{t_{\text{inacc}} + t_{\text{acc}}}, \tag{1}$$

where t_{acc} denotes time of accessibility and t_{acc} denotes time of inaccessibility (both discussed bellow).

Similarly, we may measure inaccessibility in a time period or as a function of communication requests.

$$\vartheta_m = \frac{|m| - |m_{\text{fail}}|}{|m|}, \tag{2}$$

where $|m|$ denotes the total number of messages sent and $|m_{\text{fail}}|$ the number of messages that failed to be delivered. In the following we will discuss ϑ_t while all applies equally to ϑ_m. The accessibility measure ϑ_t is symmetrical between entities A and B

$$\vartheta_t(A, B) = \vartheta_t(B, A), \tag{3}$$

while the accessibility measure ϑ_m is not necessarily symmetrical.

Indirect accessibility $\tilde{\vartheta}$ can be defined exactly in the same manner as direct, but we consider the agent to be accessible even if there is no

direct link between the agents and messages are forwarded by other agents along the path between source and destination agents. Basic properties of indirect accessibility are the same as for the direct case.

Above we presented the **agent-to-agent accessibility** between two agents – $\vartheta_m(A, B)$. This quantity is defined as in eq. 2, where the communication between the agents A and B are considered only.

More complicated is **agent-to-community accessibility**. Provided that we want to investigate agent's A accessibility in respect to a community θ (we suppose that $A \notin \theta$), the most natural way of defining the quantity is as an average of link accessibility between A and all agents in θ:

$$\vartheta_t(A, \theta) = \frac{1}{|\theta|} \sum_{B \in \theta} \vartheta_t(A, B), \tag{4}$$

Consequently, **group accessibility** in a multi-agent community is defined as

$$\vartheta_t(\theta) = \frac{1}{|\theta|} \sum_{A \in \theta} \vartheta_t \left(A, \theta \setminus \{A\}\right). \tag{5}$$

Using the probability theory, we may reason about the relation between the direct and indirect accessibility within the agent community. It holds:

$$\vartheta(\theta) \to 0 \Rightarrow \tilde{\vartheta}(\theta) \to 0 \tag{6}$$

This follows an intuitive conclusion that if agents are becoming less and less directly accessible, they can not be used to route third party traffic anymore. Such conditions, implemented in our testing scenario (see 5.1 and accessibility values in 5.2), does not allow the efficient use of middle agents. Therefore, the concept of the stand-in agent, described in section 3.2 was chosen for implementation and experiments.

As another motivation for the use of stand-in agents is a cost of forwarded communications, that may use an excessive amount of limited community resources, like battery life or usable bandwidth. The communication and social model maintenance overhead associated with forwarding management is another incentive to using the most direct communications possible. Therefore, we need to distinguish between several different accessibility situations that would provide mistakenly the same ϑ value. Let us have n agents in the community θ where 1 agent has got the *agent-to-community accessibility* quantity value $\vartheta(A_1, \theta \setminus \{A_1\}) = 0$ and $n-1$ agents have got the *agent-to-community accessibility* quantity value $\vartheta(A_n, \theta \setminus \{A_n\}) = 1$. Overall $\vartheta(\theta)$ would be $\frac{n-1}{n}$. This is the same value as in the situation where all the agents have got the same identical *agent-to-community accessibility* – $\frac{n-1}{n}$.

This is why the above suggested metrics does not distinguish between the situation where all agents are partially accessible and the situation where there is at least an agent that is totally inaccessible.

For this reason we suggest to consider the **minimal** and **maximal accessibility** in the community. These quantities are defined as follows:

$$\vartheta_t^{\min}(\theta) = \min_{A \in \theta} \vartheta_t(A, \theta \setminus \{A\}), \tag{7}$$

$$\vartheta_t^{\max}(\theta) = \max_{A \in \theta} \vartheta_t(A, \theta \setminus \{A\}). \tag{8}$$

In the best possible network environment the **accessibility is complete** – $\vartheta = 1$. This is why all the agents can freely communicate in peer-to-peer manner. Provided that $\vartheta < 1$ while still $\vartheta^{\max} = 1$ we now that there is at least one agent that can act as a facilitator and implement a communication relay. In this case we know that **accessibility is achievable**. In situations, where $\vartheta^{\min} = 0$ we know that there is at least one isolated agent that is totally inaccessible. We can say that the **accessibility is not achievable**.

The most interesting situations are outside the above listed marginal cases, e.g. when there are two teams of agents that are mutually inaccessible. For studying accessibility and inaccessibility in general the techniques from the field of graph theory need to be used.

We may consider also a very specific accessibility quantity – **accessibility distance** – denoted as $\delta_\vartheta(A, B)$, that would describe how long is the shortest path, or how many 'forwards' needs to be executed in order to convey a message between the agents A and B. Unlike the previous measures, the accessibility distance is measured in one particular moment. All edges between any two neighboring agents on this path are assumed to be accessible.

In a fully accessible environment this quantity would need to be 1 for each pair of agents. However, if for a specific pair of agent we need one middle agent (= two edges), we say that this link accessibility is 2.

We can easily see that in communities with achievable accessibility with a possible facilitator the following is true:

$$\text{if } \vartheta^{\max}(\theta) = 1 \text{ then } \forall A, B \in \theta : \delta_\vartheta(A, B) \leq 2. \tag{9}$$

The quantity $\frac{1}{\delta_\vartheta}$ captures the very important property – the difference between having a direct communication link between the agent and a need to use one facilitator different to a situation when we need instead of 8 middle-agents for sending a message between two nodes 9 middle-agents.

In our model, θ_t accessibility depends on the environment and relative agent positions only, while θ_m accessibility depends also on agent

influence factors, like communication link load or social knowledge of the agents.

3. Sharing Social Knowledge

Social knowledge represent necessary and optional information which an agent needs for its efficient operation in the multi-agent community. The social knowledge is mainly used for reduction of communication, acceleration of agents' internal reasoning processes but also for providing self-interested agents with a competitive advantage and allowing agents to reason one about the other in environments with partial accessibility. Proceeding social knowledge replaces voluminous computation between many agents. In principle there are two different concepts of manipulating the social knowledge.

3.1. Acquaintance Model

The acquaintance model is a very specific knowledge structure containing agent's social knowledge. This knowledge structure is in a fact a computational model of agents' mutual awareness. It does not need to be precise and up-to-date. Agents may use different methods and techniques for maintenance and exploitation of the acquaintance model. There have been various acquaintance models studied and developed in the multi-agent community, eg. *tri-base acquaintance model* [11] and *twin-base acquaintance model* [9]. The role of agents' social knowledge and acquaintance models in making agents' communication efficient has been studied in the research project "Multi-Agent Systems in Communication" supported by AFRL research contract F61775-99-WE099 and the concept of agents private and semi-private knowledge, that are shared and mutually maintained in agents' acquaintance models has been studied and potentials of applications in the field of OOTW coalition planning has been investigated in the research project "Multi-Agent System for Planning Humanitarian Relief Operations" supported by AFRL research contract F61775-00-WE043.

In principle, each acquaintance model is split into two parts:

- **self-knowledge** – contains information about an agent itself which shall be shared with others agents
- **social-knowledge** – contains knowledge about other members of the multi-agent system or theirs self-knowledge

While the former part of the model is maintained by the *social knowledge* provider, the latter is maintained by the *social knowledge* requestor. The knowledge that the agents are happy to share with some of the agents are denoted as **semi-private knowledge**[12]. Unlike **public knowledge**, that is shared by default by all the members of

the multi-agent community, the semi-private knowledge is shared by a specific agreement between the agents. There is also a vital part of agent knowledge equipment – agents **private knowledge**, that is not shared with any other agent. Again, there are two possible ways how the acquaintance model may be kept up-to-date.

- a **pull** model of the knowledge maintenance is often implemented by *periodical revisions* when the social knowledge requestor periodically queries the social knowledge provider for the updates of the relevant information.

- a **push** model of the knowledge maintenance can be implemented by e.g. *subscribe-advertise* protocol. The social knowledge requestor subscribes the social knowledge provider for specific information and the social knowledge provider reports on updates of the relevant information upon changes.

Social knowledge can be used for making operation of the multi-agent systems more efficient. The acquaintance model is an important source of information that would have to be communicated otherwise. Social knowledge and acquaintance models can be also used in the situations of agents' short term inaccessibility . However, the acquaintance models provide rather '*shallow*' knowledge, that does not represent a complicated dynamics of agent's decision making, future course of intentions, resource allocation or negotiation preferences. This type of information is needed for inter-agent coordination in situation with longer-term inaccessibility. We suggest introduction of the **mobile acquaintance model** concept integrated in the stand-in agent that is presented in the following.

3.2. Stand-in agent

An alternative option is to integrate the agent self-knowledge into a mobile computational entity that is constructed and maintained by the social knowledge provider. We will refer to this computational entity as a **stand-in agent**. The stand-in agent shall physically reside either on the host machine where the social knowledge requestor operates or in the safe segment of the network with guaranteed (or at least better) communication accessibility. The social knowledge requestor does not create an acquaintance model of its own. Instead of communicating with the provider, it interacts with the stand-in agent.

While usage of the classical acquaintance model is advantageous in situations where communication traffic needs to be optimized, the

stand-in agents are used primarily in the cases with partial communication accessibility.

The main philosophical difference between the concept of acquaintance model and the stand-in agent is that the stand-in agent keeps the entire copy of agents knowledge – including agents private knowledge, figure 1. This knowledge is not made available to the social knowledge requestor. However, as the stand-in agent is not a passive knowledge structure but it has got also the reasoning capability that allows to mirror the social knowledge provider interaction activity, the private knowledge can be used for certain patterns of high-level negotiation. Specifically, the stand-in agent can act on the social knowledge provider behalf, negotiate resource utilization, participate planning agent's commitments in the case of longer term inaccessibility.

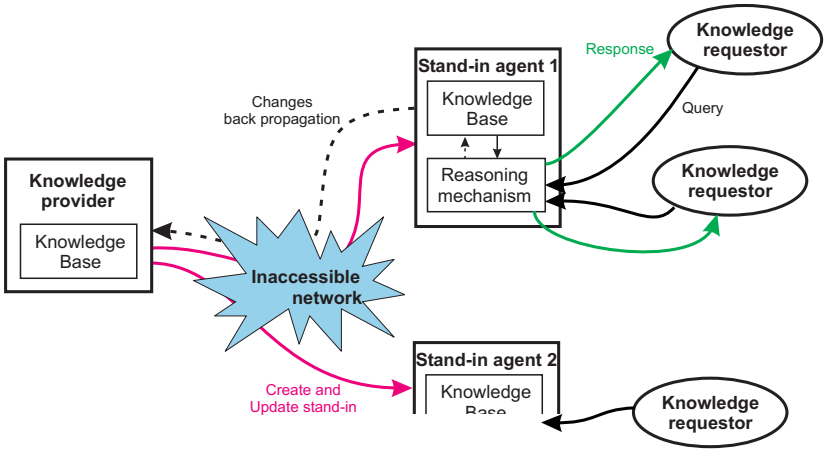

Figure 1. The concept of the stand-in agent

The stand-in agent can obviously negotiate and interact with several social knowledge requestors running on a single host or safe segment of the network. At the same time, there may be more stand-in agents representing a single social knowledge provider at different hosts or different regions of accessibility.

As a part of complex negotiation, the stand-in agent can change its self-knowledge base. These changes need to be propagated back to the self-knowledge base of the inaccessible agent. Difficulties arise in the situations when several different stand-in agents over-commit the resources of the inaccessible agent. The process of synchronization and commitments renegotiation needs to implemented in such situations.

The stand-in agent can be created permanently or temporally. Permanent stand-in is used by more clients at the same time and can also migrate inside the multi-agent system. Disposable stand-in is used only

for single transaction and is removed after completing the transaction. Depending on situation, the parent agent can have several stand-in agents simultaneously.

Stand-in agents can be used for: minimizing impact of network failure by creation of the stand-in agent at the client agent location, decreasing load of the parent agent and network link between parent and client agent.

4. Meta-Reasoning

An active and cooperative sharing of the social knowledge either within agents acquaintance models or by means of the stand-in agents is not always appropriate. There are certain aspects of agents social knowledge that cannot be simply shared, but in needs to be constructed and maintained by the knowledge owner. Instances of such information are pieces of knowledge that the agents, who provide the social knowledge, do not want to disclose, e.g. disclosure of intent, information about resources, etc. This situation may arise in the non-collaborative multi-agent systemss with agents that are trusted to various extends. In the collaborative multi-agent systemss the social knowledge can experience the 'aging process' and can become very soon outdated if the agents are inaccessible for a longer period of time.

4.1. MRinMAS Project

In either of the situations the agents need to be able to reason about the inaccessible agents. We refer to reasoning about the other agents' knowledge, mental states, resources and reasoning process as **meta-reasoning**. The concept of meta-reasoning has been thoroughly studied in research project supported by AFRL, contract Monitoring and Meta-Reasoning in Multi-Agent Systems (MRinMAS), contract number: FA8655-02-M4056.

The process of meta-reasoning in multi-agent systemss relies heavily on dividing the multi-agent community into the set of object (ordinary) agents and meta-agents. The meta-agent is an independent agent who carries out higher level reasoning process based on observing the community behavior [15] [14].

Within the MRinMAS project the abstract architecture of meta-reasoning has been designed (in terms of the model and the reasoning processes) and three separate meta-reasoning methods has been investigated:

- **automated theorem proving:** The classical theorem proving techniques based on the resolution principle has been studied. This instance of meta-reasoning is a classical deductive reasoning when all the true fact that the meta-agent assumes to be true logically follows from the set of the observation.

- **machine learning:** Version space, a classical machine learning techniques has been investigated. Version space works with the generalizations of the observed events and thus it is a clear example of inductive reasoning.

– **inductive logic programming:** A modern machine learning method that constructs a logical program that describes even small number of events. This approach is not very suitable for incremental meta-reasoning.

For specific description of the MRinMAS project results see [13].

4.2. Integration with the Project

The concept of meta-reasoning have not been directly integrated in the $\mathcal{A}$–globe platform until now for two reasons. The projects were rather short and run in parallel. It was not possible to integrate the software prototype from the MRinMAS project in a timely fashion. An important development period of this project has been devoted to implementation of the $\mathcal{A}$–globe platform.

Currently, we work on the scenario of integrating simple meta-agent in $\mathcal{A}$–globe system. Their role will be to analyze and balance the communication load between the mobile vehicles while trying to keep the confidentiality of the agents' information rather high. This is possible by trying to keep agents who are sharing semi-private information on one vehicle – container in $\mathcal{A}$–globe .

5. Experiments

5.1. Description of the experimental settings

Planning of logistic distribution of humanitarian resource is used for experiments with inaccessibility in MAS (inspired by SufferTerra scenario for humanitarian relief operation). The agents in the scenario are individuals that work together in order to achieve shared goal. Shared goal is to satisfy the requests for the delivery of commodities as soon as possible. There are three main actors:

- **Village** – a place where humanitarian resources are needed.
- **Port** – a place to which goods are coming from outside. Resources are distributed from here. It acts as aid distribution hub and can also host village stand-in agents.
- **Truck** – a vehicle capable of transporting resources and information. Each vehicle has limited load (units capacity). It can also host village stand-in agents.

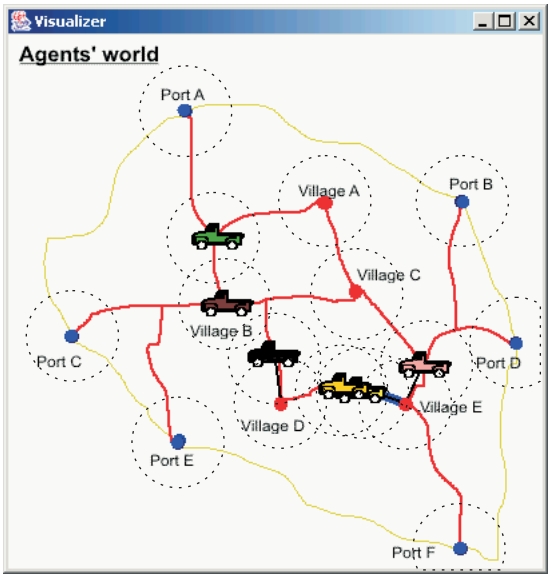

Figure 2. Agents' world scenario

Communication between actors has limited range. Communication is reliable (with no lags or dropouts) among the agents residing on one agent container, while communication across the agent containers is implemented by means of the short range radio links (for example

IEEE802.11 [6]). Therefore each agent container can communicate only with containers located within its radio range.

Specific experiment scenario is shown on figure 2. There are five villages, six ports and six trucks. Positions of villages and ports in the agents' world are chosen in such a way that no village can directly communicate with any port. They can communicate only by sending information over trucks. A truck can communicate with villages, ports and other trucks only if they are within the reach of the simulated radio link. Trucks can move on fixed routes which are predefined in configuration files. Routes of trucks for the first two scenarios, described in sections 5.1.1 and 5.1.2, are presented in the table I. There is at least one truck going from each village where some resource is needed to some port where this resource is distributed from. In other words there is no need for some middle storehouse to transfer the commodities through, but under special occasions the resource can be delivered into target village through others villages or ports.

Table I. Fixed routes of trucks for the first two scenarios

Truck	Route
Black	Port A, Village B, Village D, Village B, Port A
Brown	Port C, Village B, Village C, Village B, Port C
Green	Port E, Village B, Village A, Village B, Port E
Orange	Port B, Village E, Village D, Village E, Port B
Pink	Port D, Village E, Village C, Village E, Port D
Yellow	Port F, Village E, Village D, Village E, Port F

Routes of trucks for transloading scenario, described in section 5.1.3, are shown in the table II. Please note that the resources for villages A, C and D must be transported through villages B or E.

5.1.1. Scenario without village stand-ins

The trucks follow their routes. When some truck arrives into the village, it takes all actual requests of this village. When a truck gets into the port, it searches if there is some resource needed by some visited village and then it tries to load this resource. Resources matching the oldest requests are loaded first. Truck can carry several types of resources for different villages until its capacity is filled. Each loaded item has its own destination where it will be unloaded.

Table II. Fixed routes of transports for transloading scenario

Transport	Route
Black	Port A, Village B, Port E, Village B, Port A
Brown	Port C, Village B, Port C
Green	Village B, Village A, Village C, Village E, Village C, Village A, Village B
Orange	Port B, Village E, Port B
Pink	Port D, Village E, Port F, Village E, Port D
Yellow	Village E, Village D, Village B, Village D, Village E

5.1.2. Scenario with village stand-ins

In this scenario there are village stand-in agents in addition to villages, ports and trucks. Village stand-in agent represents interests of its mother village in a location where the village isn't accessible. These stand-ins can live on trucks or ports. When truck arrives into a village, the village scans if there is its stand-in agent on this truck. If stand-in doesn't exists, the village deploys it there and updates its knowledge. If it exists, the village only sends an update with its current knowledge, containing its current needs. In the same way the stand-ins are deployed in ports and receive updates from the stand-in on the truck container.

In a stabilized situation, when all stand-in agents were already created, the number of stand-ins on each transport equals the number of villages this transport has visited. Each port visited by any transport has at least same number of stand-ins as this transport.

These stand-ins reserve resources needed by its village before transport arrives into port. An arbitrary truck can't take the resource if it is reserved by other stand-in agent than this truck is hosting. Priority for reservation reflects the duration of request. The oldest requests are satisfied first.

Knowledge of stand-in agent is updated every time it meets mother village or any more recent stand-in of its mother village. All stand-ins of one village don't have exactly same information as its mother village at one moment. It is caused by impossibility of direct communication between all stand-ins and their mother village.

It is possible that there are delivered more resources than needed to the village. This oversupply can be loaded on a truck and redistributed to other villages, but this is not the goal in this scenario.

5.1.3. Transloading scenario with storehouses

In this case the village *request list* consists of two parts. The first part represents requests created by this village and the second part stores requests of other villages represented by stand-in agents. Each request record has destination part so it can be uniquely identified in the whole simulation world. There is a mechanism for removing already fulfilled requests so that request cannot multiply in the cycle. In other words, requests of other villages become requests of this village to other actors in the scenario.

In this scenario, resources can be distributed directly from the ports to the villages by one truck or transloaded through one or more other villages by other trucks. There are also village stand-in agents with functionality as described above in paragraph 5.1.2 in this scenario. Furthermore, the village stand-in agent can be also deployed to other villages by the stand-in agent who lives on the truck. Knowledge base of these stand-ins is updated every time when some transport with the stand-in of the same village arrives.

5.2. Experimental results

First we present measurement of accessibility in the two routes configuration in the scenarios. The table III shows accessibility matrix in the route configuration of the scenario without and with stand-in agents, sections 5.1.1 and 5.1.2. The **direct accessibility** between every two *agent containers* in the scenario, section 2.2, is measured in a time period by equation (1). There is calculated **agent container-to-community accessibility** within a community which contains all *agent containers* except the one for which this accessibility is calculated. In the right-bottom corner in the table there is **group accessibility in a multi-agent community** (equation 5). The table IV shows accessibility matrix in the route configuration of the third scenario, section 5.1.3.

Both *group accessibilities* in whole community are very close to zero. It means that we have tested the concept of stand-in agents in the highly inaccessible environment.

Five experiments with water distribution problem were done in all three scenarios. There were 49 requests generated automatically in each village from randomly pre-generated files. These requests were generated exactly in the same time of each experiment and scenario. Comparative criteria is average resource delivery time. Resource delivery time is the time measured from request creation to its complete fulfilment. Average delivery time for all scenarios is in the table V. More transparent representation of these results in the form of bar charts is

depicted in the figure 3. All these results were measured on the same computer repetitively.

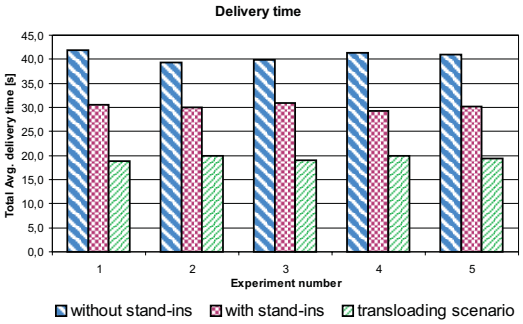

Figure 3. Average delivery time

The average delivery time in the scenario without stand-in agents is longer from 28 to 43 percent than in the scenario with stand-ins. These two scenario experiments were on the same map configuration described in paragraph 5.1.

The average delivery time in the scenario without the stand-in agents is longer from 100 to 127 percent than in this scenario. The map of the scenario is the same as in the previous experiments and the transport routes are bit different. The routes are reconfigured so that transshipping is needed, but number and speed of transports are same.

From the experiments we may identify several facts justify usefulness of the concept of stand-in agents in this scenario:

- **geographical arrangements** – usefulness of the stand-in agents depends on the position of the villages and ports. It also widely depends on the routes of the transports, their configuration and the amount. Even more specifically, the problem is very sensitive to changes of the amount and location of villages and ports with more than one route passing through them.

- **requests and resources delivery** – usefulness of stand-in agents grows when amount of resource delivery in ports is comparable or smaller than amount of resource requests in villages.

Table III. Time accessibility matrix in the route configuration of the first two scenarios

$\vartheta_t(A,B)$	Village A	Village B	Village C	Village D	Village E	Port A	Port B	Port C	Port D	Port E	Port F	Black	Brown	Green	Orange	Pink	Yellow	$\vartheta_t(A,\Theta)$
Village A	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,140**	0,000	0,000	0,000	**0,009**
Village B	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,362**	**0,363**	**0,281**	0,000	0,000	0,000	**0,063**
Village C	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,189**	0,000	0,000	**0,172**	0,000	**0,023**
Village D	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,159**	0,000	0,000	**0,148**	0,000	**0,185**	**0,031**
Village E	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,319**	**0,397**	**0,381**	**0,069**
Port A	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	**0,164**	0,000	0,000	0,000	0,000	0,000	**0,010**
Port B	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,154**	0,000	0,000	**0,010**
Port C	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	**0,209**	0,000	0,000	0,000	0,000	**0,013**
Port D	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	**0,189**	0,000	**0,012**
Port E	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	**0,156**	0,000	0,000	0,000	**0,010**
Port F	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	**0,190**	**0,012**
Black	0,000	**0,362**	0,000	**0,159**	0,000	**0,164**	0,000	0,000	0,000	0,000	0,000	**1,000**	**0,138**	**0,115**	**0,021**	0,000	**0,026**	**0,062**
Brown	0,000	**0,363**	**0,189**	0,000	0,000	0,000	0,000	**0,209**	0,000	0,000	0,000	**0,138**	**1,000**	**0,127**	0,000	**0,016**	0,000	**0,065**
Green	**0,140**	**0,281**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,156**	0,000	**0,115**	**0,127**	**1,000**	0,000	0,000	0,000	**0,051**
Orange	0,000	0,000	0,000	**0,148**	**0,319**	0,000	**0,154**	0,000	0,000	0,000	0,000	**0,021**	0,000	0,000	**1,000**	**0,235**	**0,166**	**0,065**
Pink	0,000	0,000	**0,172**	0,000	**0,397**	0,000	0,000	0,000	**0,189**	0,000	0,000	0,000	**0,016**	0,000	**0,235**	**1,000**	**0,111**	**0,070**
Yellow	0,000	0,000	0,000	**0,185**	**0,381**	0,000	0,000	0,000	0,000	0,000	**0,190**	**0,026**	0,000	0,000	**0,166**	**0,111**	**1,000**	**0,066**
$\vartheta_t(A,\Theta)$	**0,009**	**0,063**	**0,023**	**0,031**	**0,069**	**0,010**	**0,010**	**0,013**	**0,012**	**0,010**	**0,012**	**0,062**	**0,065**	**0,051**	**0,065**	**0,070**	**0,066**	**0,038**

Table IV. Time accessibility matrix in the route configuration of the third scenario

$\vartheta_t(A,B)$	Village A	Village B	Village C	Village D	Village E	Port A	Port B	Port C	Port D	Port E	Port F	Black	Brown	Green	Orange	Pink	Yellow	$\vartheta_t(A,\Theta)$
Village A	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,256**	0,000	0,000	0,000	**0,016**
Village B	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,267**	**0,342**	**0,129**	0,000	0,000	**0,273**	**0,063**
Village C	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,256**	0,000	0,000	0,000	**0,016**
Village D	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,387**	**0,024**
Village E	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,158**	**0,266**	**0,350**	**0,194**	**0,061**
Port A	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	**0,144**	0,000	0,000	0,000	0,000	0,000	**0,009**
Port B	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**0,240**	0,000	0,000	**0,015**
Port C	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	**0,363**	0,000	0,000	0,000	0,000	**0,023**
Port D	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	0,000	0,000	0,183	0,000	**0,011**
Port E	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	**0,146**	0,000	0,000	0,000	0,000	0,000	**0,009**
Port F	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	0,000	**1,000**	0,000	0,000	0,000	0,000	**0,174**	0,000	**0,011**
Black	0,000	**0,267**	0,000	0,000	0,000	**0,144**	0,000	0,000	0,000	**0,146**	0,000	**1,000**	**0,185**	**0,064**	0,000	0,000	**0,043**	**0,053**
Brown	0,000	**0,342**	0,000	0,000	0,000	0,000	0,000	**0,363**	0,000	0,000	0,000	**0,185**	**1,000**	**0,038**	0,000	0,000	**0,051**	**0,061**
Green	**0,256**	**0,129**	**0,256**	0,000	**0,158**	0,000	0,000	0,000	0,000	0,000	0,000	**0,064**	**0,038**	**1,000**	**0,090**	**0,073**	**0,046**	**0,069**
Orange	0,000	0,000	0,000	0,000	**0,266**	0,000	**0,240**	0,000	0,000	0,000	0,000	0,000	0,000	**0,090**	**1,000**	**0,206**	**0,037**	**0,053**
Pink	0,000	0,000	0,000	0,000	**0,350**	0,000	0,000	0,000	0,183	0,000	**0,174**	0,000	0,000	**0,073**	**0,206**	**1,000**	**0,034**	**0,064**
Yellow	0,000	**0,273**	0,000	**0,387**	**0,194**	0,000	0,000	0,000	0,000	0,000	0,000	**0,043**	**0,051**	**0,046**	**0,037**	**0,034**	**1,000**	**0,067**
$\vartheta_t(A,\Theta)$	**0,016**	**0,063**	**0,016**	**0,024**	**0,061**	**0,009**	**0,015**	**0,063**	**0,011**	**0,009**	**0,011**	**0,053**	**0,061**	**0,069**	**0,053**	**0,064**	**0,067**	**0,037**

Table V. Average resource delivery time

	Average delivery time [s]														
	without stand-in agents					with stand-in agents					transloading scenario				
Experiment number	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5
Village A	57,3	56,7	56,9	59,0	57,1	48,8	48,6	52,3	47,5	48,4	26,4	34,1	30,5	32,2	32,2
Village B	29,4	28,2	29,5	28,8	29,2	19,9	19,1	18,0	21,2	19,5	3,8	3,5	3,4	2,9	2,8
Village C	57,3	50,1	55,2	56,2	58,0	41,1	38,6	42,1	33,4	37,1	40,8	44,7	42,4	43,7	45,0
Village D	37,5	35,2	33,6	36,9	35,6	24,2	22,5	22,6	25,4	25,3	19,9	13,6	14,8	16,9	14,5
Village E	27,6	26,9	23,9	25,6	24,6	18,5	21,0	19,4	18,6	20,8	3,3	3,7	3,9	3,8	2,9
Total Avg. Delivery Time	41,8	39,4	39,8	41,3	40,9	30,5	30,0	30,9	29,2	30,2	18,8	19,9	19,0	19,9	19,5

6. $\mathcal{A}$–GLOBE Agent Integration Platform

This chapter explains the implementation details of $\mathcal{A}$–GLOBE. System is implemented in Java programming environment, because it is standard for Multi-Agent Systems programming and system can run on many various operating systems. The whole implementation, including experiments sources and some test agents, consists of approximately 600 classes and it is out of scope of this text to provide detailed description of platform operation. Therefore only a brief overview is given with some particulary interesting issues described in more detail. The complete source code with documentation is available on attached CD. Java documentation can be found in Sun's Java Tutorial [17], additional information in Java Forums [8]. The Java version 1.4 or higher is required to run $\mathcal{A}$–GLOBE.

6.1. Upgrade from AIT to $\mathcal{A}$–GLOBE

An Agent Inaccessibility Testbed (AIT) has been transformed into a fully functional agent interaction platform called $\mathcal{A}$–GLOBE. Parts of the $\mathcal{A}$–GLOBE systems description are identical to AIT description presented already in deliverable d.1 and/or d.2. For the sake of completeness we have decided to provide the whole description here. $\mathcal{A}$–GLOBE differs from AIT in many ways:

- **System architecture structure** – there is new system entity: *agent container*. The agent container is designed from the AIT platform packages. All static methods and variables from old platform classes have been removed to allow running several agent containers in one Java Virtual Machine. (Only one old platform can run in one JVM.) New platform provides functions for running one or more agent containers. For more details see section 6.3. This new architecture helps $\mathcal{A}$–GLOBE to be as lightweight as possible to save system resources (memory, processor time, etc.).

- **Message transport layer** – the message transport layer were optimized for speed up messages transmission. All outgoing messages can be *byte* or *XML* encoded. Byte decoder is much faster than more interoperable XML decoder.

- **Environment Simulator** – there are two `gis` services: `master` and `client`. These services bind agent containers together into one system with central ES agent which simulates environment (controls position and accessibility) of an agent containers. Environment Simulator is designed as an ordinary agent which is

connected to the `gis/master` service. It is easy to create new Environment Simulator Agent using `gis` services.

- **Sniffer Agent** – it is on-line tool for monitoring messages and their transmission status in the distributed multi-agent system. It allows investigate all messages from whole system at the one place. It is mainly used during development, debugging and integration phase.

6.2. Functional Description

$\mathcal{A}$–GLOBE is an agent interaction platform. It was designed for testing experimental scenarios featuring position and inaccessibility. But $\mathcal{A}$–GLOBE can be used as a simple platform without these extended functions. The platform provides functions for the residing agents, such as communication infrastructure, store, directory services, migration function, deploy service, etc. It is very fast and relatively lightweight. Comparison to the others agent platforms can be found in section 7.

If $\mathcal{A}$–GLOBE is started in extended version with `gis` services and Environment Simulator agent, it is most suitable for experimental scenarios including both static (e.g. towns, ports, etc.) and mobile units (e.g. vehicles). Such scenario is defined by a set of actors represented by agents residing in the agent containers. The ES agent simulates dynamics (physical location, movement in time and others prameters) of each agent container connected together by `gis` services. The ES agent can also controls accessibility among all agent containers. The client side of `gis` services applies accessibility restriction in the message transport layer of the agent container.

6.3. System Architecture

The $\mathcal{A}$–GLOBE schema design is shown in figure 4, it consists of several components:

- **platform** – provides basic functions for running one or more agent containers, such as container manager, library manager (section 6.4);
- **agent container** – skeleton entity of $\mathcal{A}$–GLOBE, provides basic functions, communication infrastructure and storage for agents (section 6.5);
- **services** – provides some common functions for all agents in one container (section 6.6);

– **environment simulator (ES) agent** – simulates real world environment and controls visibility between others agent platforms (section 6.7);

– **scenario agents** – represents actors in the scenario.

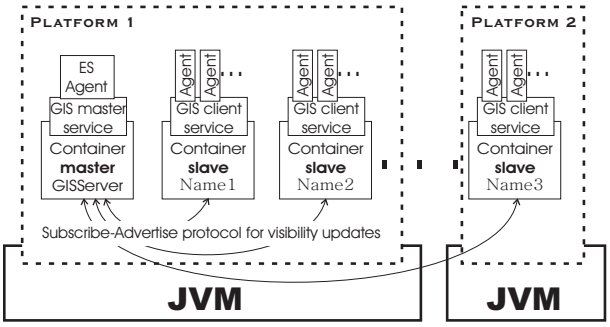

Figure 4. System Architecture Structure

Security: Security issues are not explicitly addressed in the platform design. This has two reasons. Firstly, the platform is meant for experiments with inaccessibility and these experiments currently put no requirements on security. Secondly, certain level of security could be provided by the underlying network level. However, if any security requirements arise later, the platform can be extended.

All the platform components described later have a main class with corresponding name, but the whole component functionality is usually provided by a set of classes. The italic font (*Agent Manager*) is used when referring to the whole component, while typewriter font (`AgentManager`) refers to the class.

6.4. Agent Platform

The main design goals were to create the platform as lightweight as possible and to make the platform easily portable to another machine. The platform is implemented as an application running on Java Virtual Machine (JVM). Several platforms can run simultaneously (maximum 1000), each in its own JVM instance. When new agent container is started, it can be specified in which platform container will be created and running (see table VII).

The platform has two components:

– **Container manager.** The Container Manager takes care of starting, execution and finishing *agent containers*.

– **Library Manager.** The Library Manager obtains libraries required by agents and services and maintains them. The Library Manager is also responsible for moving libraries of any migrating agent to other platforms where required libraries are not found.

In the one *agent platform* can run one or more *agent containers*. Containers are mutually independent except for shared common *library manager*. Usage of one *agent platform* for all *containers* running on one computer machine is beneficial because it rapidly decrease system resources requirements (use of single JVM), e.g. memory, processor time, etc.

The *library manager* is part of *agent platform* because all classes inside one Sun's Java Virtual Machine use same class loader. The Agent Platform is also responsible for starting new container in the existing platform with the same number. New platform is started if there is request for starting *agent container* on agent platform which does not exists yet. If there isn't any container in a platform, the platform is automatically shut down. The Agent Container is started through the main class of agent platform `ait.platform.Platform`. Starting arguments are described in the section 6.5.

6.4.1. Library Manager

The *Library Manager* takes care of the libraries installed in the platform and monitors which agents/services use which library. Descriptor of each agent and service specifies which libraries the agent/service requires. Whenever an agent migrates or agent/service is deployed, the *Library Manager* checks which libraries are missing on the platform and obtains them from the source platform. The inter-platform functionality of *Library Manager* is realized though the service `library/loader` (this service is present on every agent container and consists of two classes `LibrarySender` and `LibraryRequester`). The UML library deployment sequence diagram is shown on figure 5.

The *Library Manager* provides `ClassLoader` which allows the JVM to load classes from platform registered libraries. `AgentManager` and `ServiceManager` both use this `ClassLoader` when creating agent/service objects. And the `MessageTransport` uses this when unmarshaling message content.

The user can add, remove and inspect libraries using the GUI. The GUI library information is shown on figure 6. Each library in the platform is described by the library descriptor (`TheLibrary`), whose structure is shown in table VI.

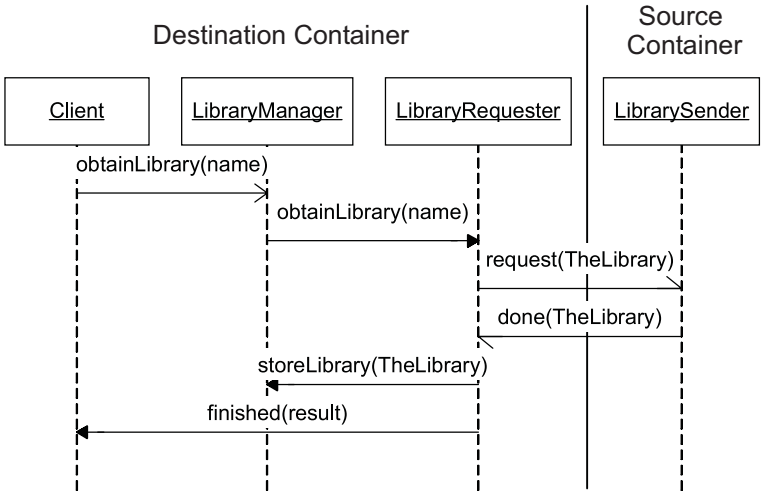

Figure 5. Library Deployment Sequence Diagram

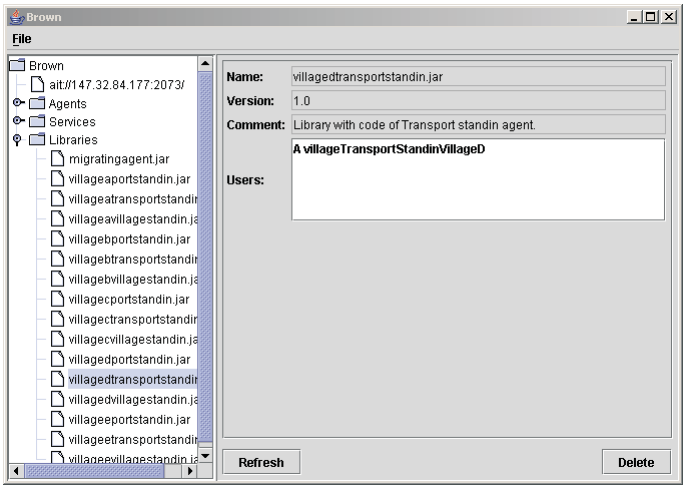

Figure 6. GUI: Library Information

6.5. Agent container

The agents are not stand-alone applications but objects inside the agent containers. Each agent is executed in a separate thread. The schema of general agent container structure is shown on figure 7. Most of the higher level container functionality (agent deployment, migration, directory, etc.) is provided as standard container services (see 6.6).

The agent container components are:

– **Container Core.** The Container Core starts and shuts down all container components.

Table VI. `TheLibrary` Fields

Field Name	Car.	Comment
`Name`	1	Library name `(something.jar)`
`Version`	1 or 1	Library version
`Comment`	0 or 1	The service description
`Data`	0 or 1	Base64 encoded `.jar` file

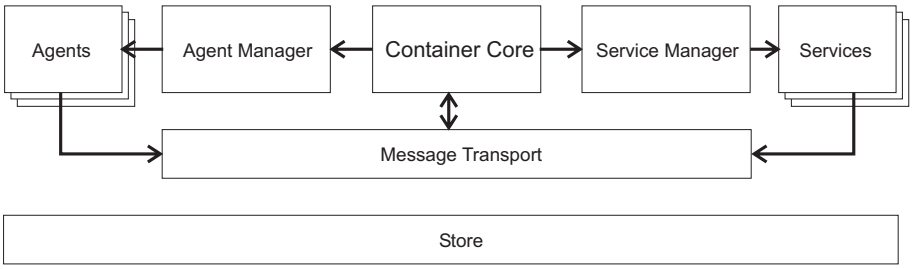

Figure 7. The Agent Container Structure

- **Store.** The Store provides persistent data storage facility. It is used by all container components, agents and services.
- **Message Transport.** The Message Transport is responsible for sending and receiving messages from and to the container.
- **Agent Manager.** The Agent Manager takes care of creation, execution and removal of agents.
- **Service Manager.** The Service Manager takes care of starting and stopping the services and their interfacing to other container components.

The *agent container* is started through main platform class `ait.platform.Platform`. The *Agent Platform* is responsible for starting or attaching new containers, see section 6.4. There are several starting arguments described in the table VII.

There is one mandatory starting argument *container name*. The *container name* must be unique inside one system build from several containers. This *name* is also used for determination specific store subdirectory for the *agent container* and registering to *Environment Simulator Agent*. When port number isn't provided, the agent container automatically selects first free port in the system. The store root directory needs to point into directory where this process has read and write privileges. If neither *master* nor *slave* argument isn't used,

Table VII. Agent Platform starting arguments

Attribute	Mandatory	Description
`-name name`	Yes	Set the 'name' as container name
`-port nnnn`	No	Container will listen on port nnnn
`-platform nnn`	No	Create in or attach new container to platform `nnn` (0-999)
`-gui`	No	Show container GUI after starting
`-root dir`	No	Select other store root directory than default
`-p name=value`	0 or more	Set special parameter 'name' to 'value'
`-master`	No	Start as *gis master* container; Cannot be used with parameter `slave`
`-slave`	No	Start as *gis slave* container; Cannot be used with parameter `master`
`-masterAddress` ait://host:port	No	Use this address for registering in the master container
`-XMLmessages`	No	All messages will be encoded into XML

the agent container is started as stand-alone environment for running agents. These two arguments are used if *Environment Simulator* or other position functions is needed, see figure 4. If both arguments *master* and *slave* are used at once, the *master* has priority. The *Master Address* of master container is supposed to be in special addressing format and is used only when container is started in *slave mode*. If *master address* isn't provided, the agent container automatically tries to find *master container* in the local network. If *master container* isn't found, default address: `ait://127.0.0.1:1024` is used.

6.5.1. Container GUI

The *agent container* has a graphic user interface (GUI), which gives the user an easy way to inspect container state and to install or remove its components. The GUI could be shown or hidden both locally and remotely (by message). The GUI screen shot is shown in figure 8.

The platform window has two parts. The tree on the left side shows names of agents, services and libraries present on the container. The right side shows detailed information about the object selected in the tree. Besides displaying this information, the GUI lets the user to install and remove agents, services and libraries.

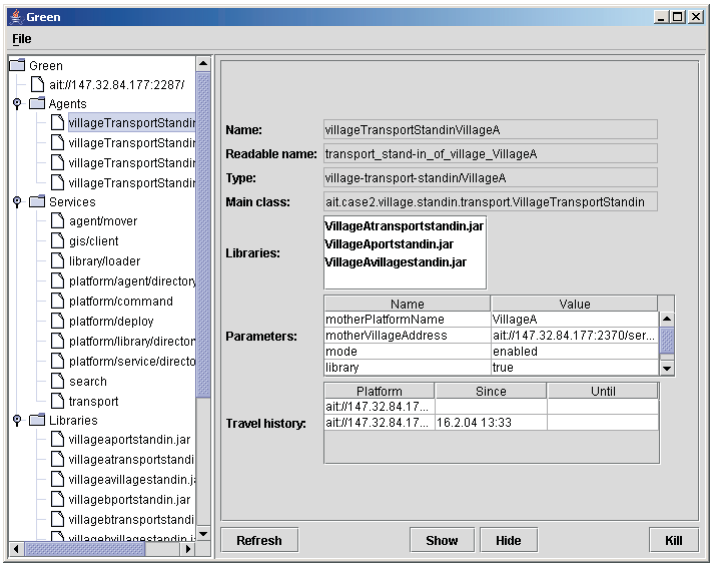

Figure 8. Container GUI: Agent Information

Moreover, the agents and services are allowed to create their own GUI without any restrictions. This increases an impact of agent failure to the rest of the agent container, but highly improves the usability.

6.5.2. Message Transport

The *Message Transport* is responsible for sending and receiving messages. Shared TCP/IP connection for message sending is created between every two platforms when the first message is exchanged between them. The message flow inside the platform is shown on figure 9.

The message structure respects FIPA-ACL [3]. Message fields are shown in table VIII. The message are encoded in eXtensible Markup Language (XML, [2]) or byte encoded depending on occurrence of container starting attribute *XMLmessages* (section 6.5). The structure of each XML document is described by Document Type Definition (DTD, [1]) file. For coding (marshaling) and decoding (unmarshaling) XML documents the Java APIs for XML Binding (JAXB, [7]) package is used. The JAXB[2] is a useful tool which is able to automatically generate Java classes from DTD file and binding script. Such generated classes have ability to marshal and unmarshal themselves and provide access methods to their fields.

[2] JAXB version 1.0 Early Access was used. Currently the version 1.0 Beta is available, which uses XML Schema instead of DTD for classes generation.

[3] Car. = Cardinality; number of occurrences of the element.

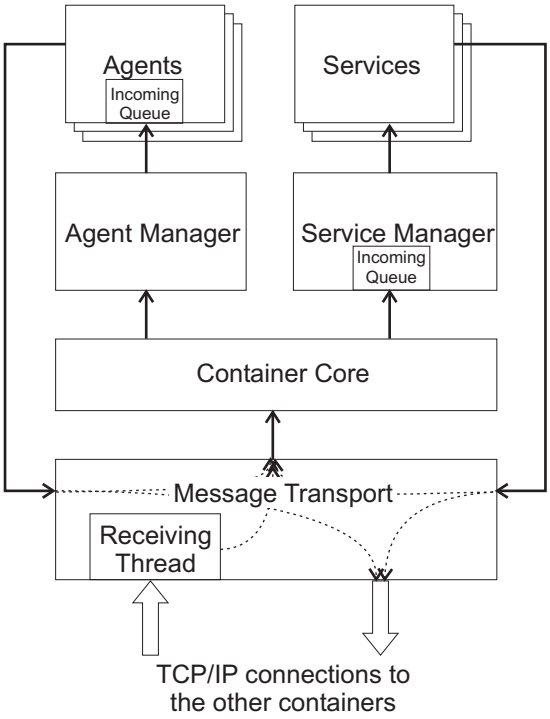

Figure 9. Message Flow

The `Content` of the message is transformed into XML in two ways: if the `Content` object is marshalable, its `marshal` method is used, otherwise its `toString` method is used. And vice versa, during the message unmarshal process, an attempt to unmarshal the `Content` into an object of type specified by `Ontology` is made. When the attempt fails, the `Content` is treated as `String`. This method allows the agents/services to easily pass marshalable object through messages.

For transport, all binary data are Base64 encoded using open source Base64 coding and decoding routines by Robert Harder [10].

Class `ait.platform.agent.Agent` provides two methods for sending and receiving messages:

```
1 public void sendMessage(Message m);
2 protected void handleIncommingMessage(Message m);
```

And class `ait.platform.service.Service` has:

```
1 public void sendMessage(Message m);
2 public void incommingMessage(Message m);
```

If message cannot be delivered, method `sendMessage` throws exception `InvisiblePlatformException`.

Table VIII. Message Fields

Field Name	Car.[3]	Comment
`Sender`	1	Sender address
`Receiver`	1	Receiver address
`Performative`	1	Message performative, see FIPA Performatives [4]
`Content`	0 or 1	Content (XML encoded)
`Protocol`	0 or 1	Conversation protocol, see FIPA Protocols [5]
`Ontology`	0 or 1	Content ontology
`ConversationID`	0 or 1	ID for message matching
`InReplyTo`	0 or 1	`ReplyWith` from preceding message
`ReplyWith`	0 or 1	ID to copy to `InReplyTo`
`Reason`	0 or 1	Human readable debugging info

Naming and Addressing: An address has the following syntax:

`ait://platform_ip:port/[agent|service]/name`.

The `platform_ip` is the IP address of machine running the platform, `port` is the TCP/IP port on which the specific *agent container* is listening. The agent name is globally unique and is normally generated by platform during agent creation. The service name is unique only within one agent container (services cannot migrate) and is specified by the service creator. The names are case insensitive and consist of alphabetic and numeric characters. The agents and service names can also contain the slash `'/'` for better name structure (ex. `ait://127.0.0.1:1024/service/my/service`).

6.5.3. Store

The purpose of *Store* is to provide permanent storage through interface which shields its users from the operating system filesystem. Each entity in the agent container (agent, service, container components) is assigned its own virtual storage, which is unaffected by the others. Whenever an agent migrates, its store content is compressed and sent to the new location. The storage is able to store types `boolean`, `int`, `String`, serializable objects, marshalable objects[4] and files. The stored elements are identified by `String` name. Internally, all elements are stored as files.

The *agent container* store root resides in subdirectory `.platform` of the current user home directory (see table IX) if no other store root subdirectory is specified as argument to platform at starting time. Each *agent container* running on the system has its own subdirectory with

[4] The marshalable objects are described in section 6.5.2 – Message Transport.

Table IX. User Home Directories

Directory	System
`~`	Unix, Linux
`C:\Windows`	Windows 95/98
`%HOMEPATH%`	Windows 2000/XP

structure shown in table X. When a container with a given name is run for the first time, new storage is automatically created and filled by default values.

Table X. Agent Container Store Structure

Directory	Usage
`container_name`	Container store root
`/agents`	Agent stores
`/services`	Service stores
`/libraies`	Libraries
`/prefs`	Container settings

Manipulation with the store is done by using method of `ait.container.Store` class. Agent or service gets `Store` instance by using methods of `ait.container.AgentContainer` obtained by calling agent or service method `getContainer()`:

```
1 public Store getAgentStore(String agentname);
2 public Store getServiceStore(String servicename);
```

Once having the right `Store` instance, the data can be easily stored and retrieved by bunch of `putXXX` and `getXXX` methods. The `getXXX` methods for simple types require parameter specifying default value for the entry. Entries that are not needed any more can be deleted by using `public void deleteKey(String key)` method. Existence of entry can be verified by `public boolean exist(String key)` method.

6.5.4. Agent Manager

The *Agent Manager* takes care of agents running on the agent container. It creates agents, re-creates them after platform restart, routes the incoming messages to the agents, packs the agents for migration and removes agent's traces when it migrates out of the platform or dies.

The user can create, kill and inspect agents using the GUI. The agent GUI information is shown on figure 8.

Table XI. `AgentInfo` Fields

Field Name	Car.	Comment
`Name`	1	Globally unique name
`ReadableName`	1	User friendly name
`Type`	1	Agent type (any string)
`MainClass`	1	Agent's main class
`TravelHistory`	0 or more	Agent's migration history
`Platform`	1	Agent container address
`Start`	0 or 1	Start time stamp
`Stop`	0 or 1	Stop time stamp
`Param`	0 or 1	Parameters to pass to agent
`Name`	1	Parameter name
`Value`	0 or 1	Parameter value
`Libraries`	1	Libraries required
`Library`	0 or more	Library name
`Data`	0 or 1	Base64 encoded agent `Store`
`Serialized`	0 or 1	Base64 encoded serialized agent

The structure of `AgentInfo` (agent descriptor) is shown in table XI. The `AgentInfo` is used for agent migration, deployment and describes agents currently resident in the agent container (with some fields left blank).

6.5.5. Agents

The agent container hosts two types of entities that are able to send and receive messages: agents and services. While the first ones are the reason of $\mathcal{A}$–globe existence the latter ones are meant as an extension of the container infrastructure.

The agents are autonomous entities with unique name and ability to migrate. There is a separate thread created for each agent. A wrapper running in the thread executes the agent body. Whenever an unhandled exception is thrown by the agent or agent body exits, the

control is passed back to the wrapper, which handles the situation. Therefore potential agent failures are not propagated to the rest of the agent container. The return value of the agent main method is used to determine agent's termination type (die, migrate, suspend). Agents could be deployed to remote container.

Every agent is extended from main agent class `ait.container.agent.Agent`. The class provides basic functions for an agent, such as:

- `getContainer()` – returns container instance,
- `getName()` – returns name of the agent,
- `migrate(Address destination)` – start migration procedure from agent's will (section 6.5.5.1),
- `sendMessage(Message m)` – this method is used for sending message by the agent.

If an agent wants receive messages without using conversation discrimination (all incoming messages to this agent go to one method), the agent must overload method `handleIncommingMessage(Message m)` otherwise it must use **conversation manager** with tasks, described in the section 6.5.5.2.

6.5.5.1. Migration Procedure

In order to successfully migrate, the agent has to support serialization. The serialization is an instrument of the Java language to store objects into a sequence of bytes. The object serialization process is almost automatic, but the object has to provide a special support for some special cases.

The migration sequence is shown on figure 10. All exceptions that might occur during the process are properly handled and the communication is secured by timeouts. If the migration cannot be finished for any reason, the agent is re-created in its original container.

If the `done` message is successfully sent by the agent destination container but never received by the source container, two copies of the agent emerge. If the `done` message is received by the source container, but the agent creation fails at the destination container, the agent is lost. These events can never be fully eliminated due to different inaccessibility types (section 2.1), but maximum caution was given to minimize their probability.

For the migration process class `ait.container.agent.Agent` provides a method `protected void migrate(Address destination)`. If some agent wants to migrate of its own will, it calls this method

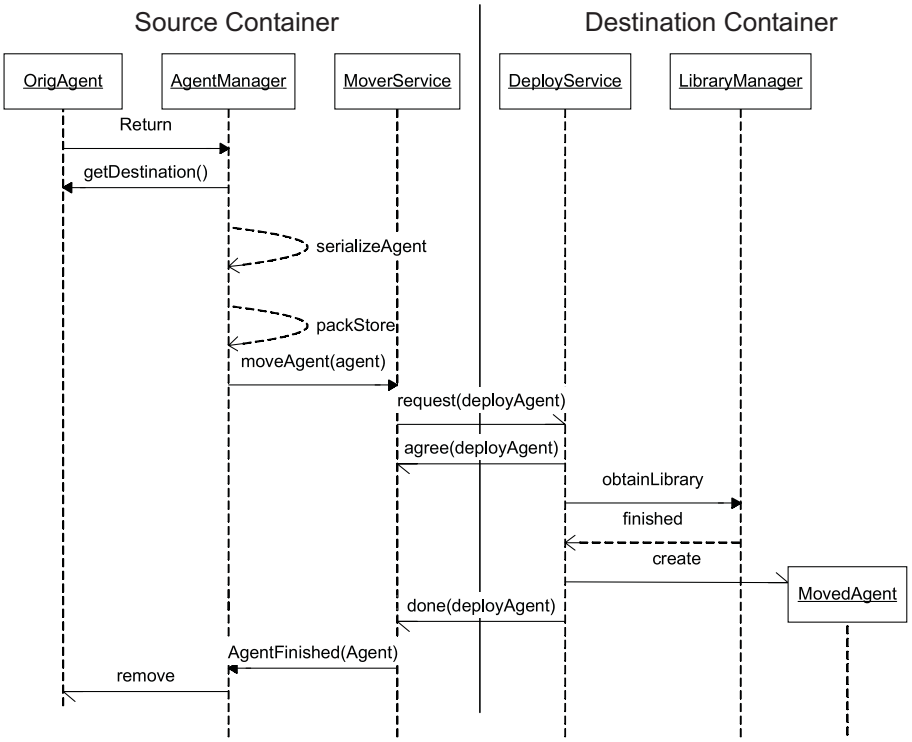

Figure 10. Agent Migration

with destination address. The migration process was tested and works smoothly also between two containers hosted on different operating systems.

6.5.5.2. Conversation Manager and Tasks
Usually, an agents deals with multiple jobs simultaneously. To simplify a development of such agents, the $\mathcal{A}$–GLOBE offers tasks. A task is a class extended from class `ait.container.Task`. A task is able to send and receive messages and to interact with other tasks. The `ConversationManager` takes care of every message received by the agent to be routed to the proper task. The decision, to which `Task` a message should be routed, depends on the massage `ConversationID`. The `ConversationID` should be viewed as a 'reference number'.

If we compare an agent to an institution, then a `Task` would be dedicated office worker. The institution deals with lots of agendas, but each worker deals only with his current case. When external entities interact with the institution, they mention the reference number in every letter. Based on the reference number the messenger boy delivers the letter to the appropriate worker.

To switch on the task support, agent's method for starting the conversation manager `startConversationManager()` has to be called. After calling this method, the message events are no more routed to agent's method `handleIncommingMessage`, but to the `ConversationManager`, who forwards them to proper task's `handleIncommingMessage` method. The conversation manager can be accessed by agent's member variable `cm`. There are two ways to associate the particular task with some `ConversationID`:

- When a task sends out a message using it's `sendMessage`, it becomes associated with the message `ConversationID` (the `ConversationID` of outgoing messages is automatically filled in by the conversation manager).
- A task can be associated with `ConversationID` manually by calling conversation's manager `registerTask(String convID, Task t)` method.

All messages that do not have the `ConversationID` associated with the particular task are routed to so called idle task. The idle task is set by the `setIdleTask(Task t)` method of conversation manager.

When a task is finished, it should deregister from the conversation manager by calling the `cancelTask()` method.

To offer the tasks easy way to implement timeouts and to save system resources, the `AgentContainer` class contains public final variable `TIMER` (`java.util.Timer`).

6.5.6. Service Manager

The *Service Manager* takes care of services present in the agent container. The user can start, stop and inspect the services using GUI. The GUI service information is shown on figure 11.

There are two types of services – user services and system services. The system services are automatically started by the container and form a part of the container infrastructure (agent mover, library deployer, directory services etc.). The system services cannot be removed. The user services can be started by user or any agent/service. The user services can be either permanent (started during every container startup) or temporary (started and stopped by some agent).

The structure of `ServiceInfo` (service descriptor) used for service deployment is shown in the table XII.

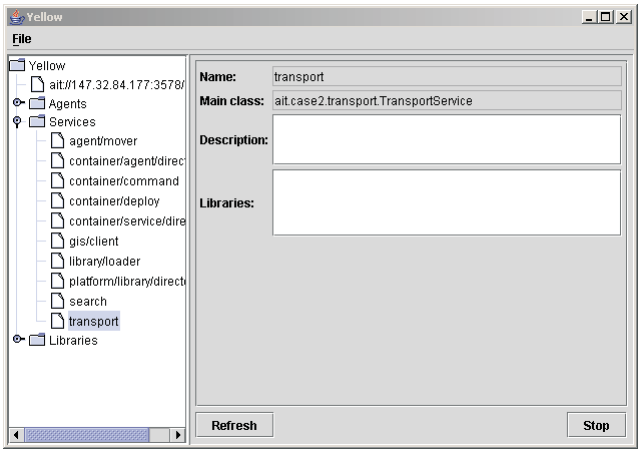

Figure 11. GUI: Service Information

Table XII. `ServiceInfo` Fields

Field Name	Car.	Comment
`Name`	1	Platform unique name
`MainClass`	1	Service's main class
`Libraries`	1	Libraries required
`Library`	0 or more	Library name
`Data`	0 or 1	Base64 encoded service `Store`
`Description`	0 or 1	The service description

6.6. Services

The services are bound to particular container by their identifier. There could be the same service on several containers. The services do not have their own dedicated thread and are expected to behave reactively on response to incoming messages and function calls. Services can be deployed to remote container.

The agents (and services or container components) have two ways to communicate with a service. Either via normal messages or by using the service shell. The service shell is a special proxy object that interfaces service functions to a client. The UML sequence diagram of service shell creation and use is shown on figure 12.

The advantage of service shell is an easy agent migration (for migration description see section 6.5.5.1): while the service itself is not serializable, the service shell is. When an agent migrates, the shell serializes itself with information what service name it was connected to. When the agent deserializes at the new location, the shell reconnects

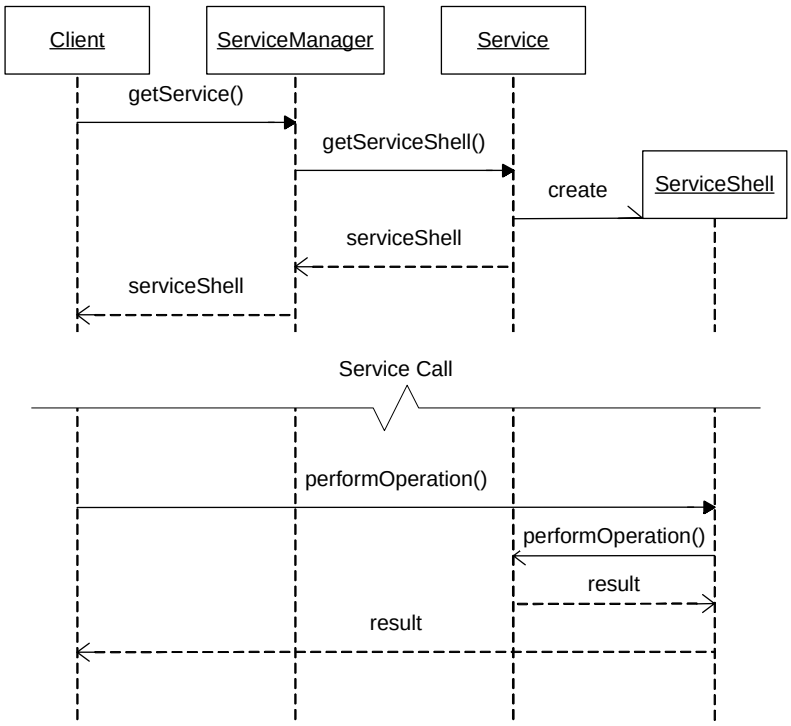

Figure 12. Service Shell Operation

to its service at the new location. Before using the service the agent should call shell's `isValid` method to find out whether it is connected to the service.

When a service is shut down, it notifies it's shells so that they refuse subsequent service calls.

There are several common services described in the table XIII. These services are automatically started by the agent container and provide common functions for all agents. Last two services `gis/master` and `gis/client` depending on starting arguments `master` and `slave`, see section 6.5.

6.7. Environment Simulator

The purpose of ES is to simulate the real world environment. More specifically the ES models the platform mutual accessibility described by equation (1) and informs each container about other platforms inside its communication range. Besides visibility, the ES can inform the containers about any other parameters (eg. position, temperature, ...). The ES consists of two parts: the **ES Agent**, which generates the information and **gis services** that are present at every platform

Table XIII. System services description

Service name	Description
`container/command`	Service through which container core remotely receives commands (show/hide GUI, shutdown)
`container/service/directory`	Provides searching of service addresses matching some search criteria
`container/agent/directory`	Provides searching of agent addresses matching some search criteria
`platform/library/directory`	Provides searching of library matching some search criteria
`container/deploy`	Service responsible for starting an agent from agent info record (table XI)
`gis/master`	Master side of Environment Simulator service
`gis/client`	Client side of Environment Simulator service

connected in the scenario, figure 4. Two types of the ES Agent can be easily started from main menu of the master container GUI. Master container is started with the attribute `master`.

6.7.1. Environment Simulator Agent

The Environment Simulator (**ES**) is implemented as an ordinary Agent. However, the agent container hosting the ES is special, because it does start the `gis/master` service only. Absence of the `gis/client` service allows the container freely communicate (perfect accessibility, section 2.2) with all other containers. The structure of data sent by the server to clients is shown in table XIV.

Table XIV. `GISInfo` Fields

Field Name	Car.	Comment
`VisibleContainer`	0 or more	Containers in radio range
`Name`	1	Container name
`Host`	1	Container host address
`Port`	1	Containers communication port
`Param`	0 or 1	Environment parameters
`Name`	1	Parameter name
`Value`	0 or 1	Parameter value

The ES agent architecture allows simulation of complicated platform motion, suited exactly for experiments performed. There are two ES agents implemented and described in next subsections.

6.7.1.1. Matrix ES Agent
The Matrix ES Agent implementation provides simple user-checkable visibility matrix, as shown on figure 13. This implementation does not need any other configuration. When an `gis/client` service subscribes to the `gis/master` service, the container name is automatically added to the matrix, creating a new column and new row. The user simply check which containers can communicate together and which can't.

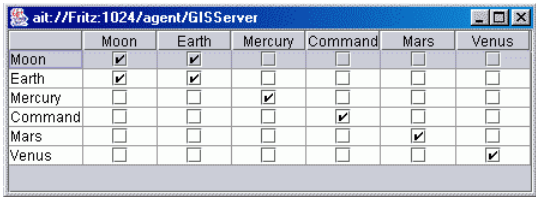

Figure 13. Platform Visibility Matrix

6.7.1.2. Graphics ES Agent
The Graphics ES Agent is fully automatic environment simulator. It moves mobile agent containers representing mobile units in virtual world and automatically controls accessibility between them. The visibility is controlled by means of simulation of the short range wireless link. Therefore each container can communicate only with containers located inside predefined radius limit. As the containers move, connections are dynamically established and lost. The Graphics ES Agent GUI has a control window, shown on figure 14, and a visualizer window, shown on figure 2. Through the control window user can load configuration from file, start and stop simulation or can send control messages to platforms.

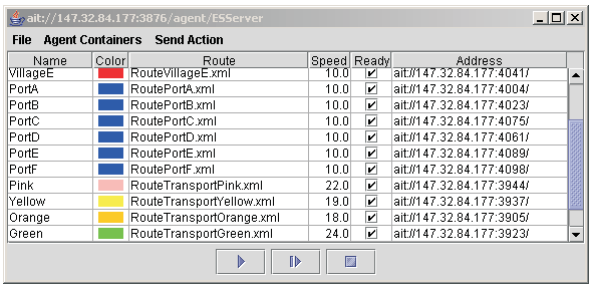

Figure 14. Control window of ES server

Number of agent containers and other parameters are read from main XML configuration file called `default.xml`. The structure of this file is shown in table XV. And the structure of map definition file is shown in table XVI.

Table XV. The structure of `default.xml`

Field Name	Car.	Comment
`Name`	1	Configuration name
`Map`	1	Reference to map XML definition file
`Range`	0 or 1	Wireless range limit
`VisParam`	0 or more	Visualization parameters
`Name`	1	Parameter name
`Value`	1	Parameter value
`Container`	1 or more	Client containers
`Name`	1	Client container name
`Route`	1	Reference to route XML file
`Color`	0 or 1	Color of the container in visualizer
`Speed`	0 or 1	Speed of the container
`Icon`	0 or 1	Reference to icon file

Table XVI. The structure of map file

Field Name	Car.	Comment
`Location`	0 or more	Location in the map
`Name`	1	Name of location
`PosX`	1	X position of location on the map
`PosY`	1	Y position of location on the map
`isCity`	true or false	Define if the location is a city

Agent containers move on fixed routes listed in the main configuration file. Each container route is stored in separate file. The structure of this file is shown in table XVII. If `LoopingType` is set to `loop`, the route of the container continues after last segment again from first segment in the loop. `Bounce` type means that the container bounce on the segments between first and last point. The route can contain pause or route events. The route events are used for noticing the **gis/client** that it arrives somewhere, and movement can be paused until the client sends a command to go on.

Table XVII. The structure of route definition file

Field Name	Car.	Comment
`LoopingType`	loop or bounce	Looping type of this route sequence
`StartX`	0 or 1	Starting X position
`StartY`	0 or 1	Starting Y position
`StartLoc`	implied	Or starting location name
`RouteSegment`	0 or more	One segment of the route
`Type`	line or curve	Segment type
`X`	0 or 1	End point - X coordinate
`Y`	0 or 1	End point - Y coordinate
`LocName`	implied	Or end point location name
`RouteWait`	0 or more	Waiting event
`WaitTime`	1	Waiting duration in msec
`RouteEvent`	0 or more	Route event
`PostEvent`	0 or 1	Send this event to **gis/client**
`WaitEvent`	0 or 1	Wait until receive this event from **gis/client**

6.7.1.3. GIS services
Gis services distribute visibility information to all container message transport layers. There are two types of gis services. One for sever side and one for client side, as presented on figure 4.

If the container is started with the attribute `master` (see starting attributes in section 6.5), the server service named `gis/master` is automatically started by the container core. The service starts listening for broadcast UDP packet with auto-configuration request. The service manages list of logged containers. The Environment Simulator Agent doesn't communicate directly with logged containers. It uses service shell of the `gis/master` for every communication. It is therefore easy to create a new Environment Simulator Agent with new functionality that will be use this shell for communication with all clients.

The client service named `gis/client` is started by the container core if the container is started with the attribute *slave*. After the container startup, the service subscribes with the `gis/master` to receive the environmental updates. The address of server container can be passed to the container as startup attribute. Otherwise client tries to find server automatically in the local network by sending broadcast packets. If automatic detection fails, the client uses localhost address with port 1024 for connecting to the server. Any container component (agent or service) wishing to receive the environment updates (visibility and other pa-

rameters) can register as `gis/client` service listener (must implement interface `ait.gis.service.client.GISVisibilityListener`).

The most important of `gis/client` service listeners is the `MessageTransport`, which needs the update information to apply visibility restrictions. Before receiving first update from the `gis/client` service, the `MessageTransport` sends messages to any platform. As soon as visible platforms list is available, the messages to platforms not present on the list are returned as undeliverable. Therefore, if no ES Agent is started, all platforms are connected without any restrictions.

6.8. Sniffer Agent

The **Sniffer Agent** is an on-line tool for monitoring all messages and their transmission status (delivered or invisible target). This tool helps to find and resolve communication problem in multi-agent systems during development phase in the $\mathcal{A}$–globe.

The sniffer can be started only on an agent container where **gis/-master** service is running through master container gui main menu. After sniffer start, all messages between agents and services inside any container or among two agent containers are monitored. Messages can be filtered by the sender or receiver of the message. All messages matching the user-defined criteria are shown in the sniffer gui, as shown on figure 15. The message transmission status is emphasized by type of line. The color of the message correspond to the message performative. By double clicking full message details are shown 16.

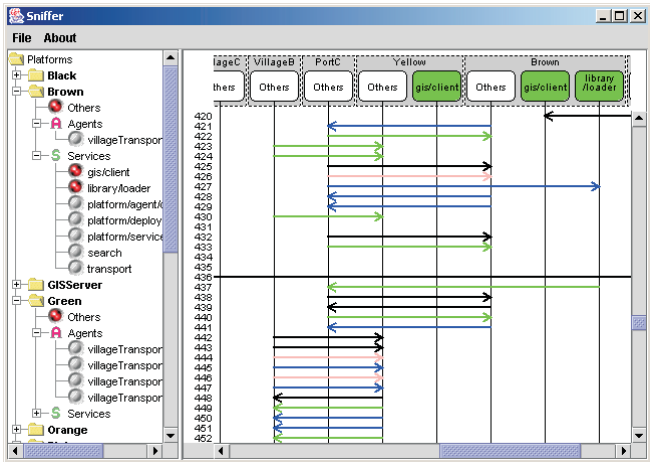

Figure 15. The sniffer GUI

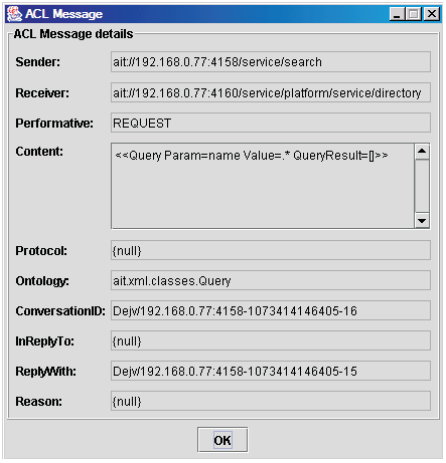

Figure 16. The message detail window

7. Platform comparison

This section presents the results of comparison of available JAVA-based agent development frameworks (agent integration platforms) evaluated by an independent expert Pavel Vrba from Rockwell Automation Research Center Prague [16], which were carried out in a cooperation with the Gerstner Laboratory. Authors wish to express acknowledgement to Rockwell Automation Research Center Prague for mutually beneficial cooperation in the platform evaluation process.

Firstly, the particular benchmark criteria, which the agent platform should provide are identified (e.g. small memory footprint and message sending speed). For selected agent platforms - JADE, FIPA-OS, ZEUS, JACK, GRASSHOPPER and $\mathcal{A}$–GLOBE - the results of benchmarking are presented with respect to message sending velocity. This property is a crucial property in many applications.

7.1. Agent platforms characteristics

The agent development tool, often called an *agent platform*, provides the user with a set of JAVA libraries for specification of user agent classes with specific attributes and behaviors.

In many applications there are specific requirements on the properties of the agent platform:

– **FIPA compliancy** – compliancy with the FIPA standards is a crucial property ensuring the interoperability. FIPA specifications address several aspects of agent platforms, not only the *inter-agent communication* that follows the Agent Communication Language.

How the agents should be organized and managed within the agent community is covered by the *agent management* specifications. The FIPA specification of the *message transport protocol* (MTP) defines how the messages should be deliverd among agents within the same agent community and particularly between different communities.

- **memory requirements** – This issue is mainly interesting for deploying agents on small devices like mobile phones or personal digital assistants (PDAs) which can have only a few megabytes of memory available. This issue is also important for running thousands of agents on the one computer at the same time.

- **message sending speed** – for many applications the message sending speed between agents is crucial parameter of the agent platform. The agent platform runtime, carrying out interactions, should be fast enough to ensure reasonable message delivery times.

7.2. Message speed benchmarks

The selected platforms have been put through a series of tests where the message delivery times have been observed under different conditions.

In each test, so called *average roundtrip time* (avgRTT) is measured. This is the time period needed for a pair of agents (let say A and B) to send a message from A to B and get reply from B to A. JAVA `System.currentTimeMillis()` method is used for measuring time which returns the current time as the number of milliseconds since midnight, January 1, 1970. The roundtrip time is computed by the agent A when a reply from B is received as a difference between the receive time and the send time. An issue is that a millisecond precision cannot be mostly reached; the time-grain is mostly 10ms or 15ms (depending on the hardware configuration and the operating system). However it can easily be solved by repeating a message exchange several times (1000 times in our testing) and computing the average from all the trials.

As can be seen in table XVIII, three different numbers of agent pairs have been considered: 1 agent pair (A-B) with 1000 messages exchanged, 10 agent pairs (A_1-B_1, A_2-B_2, ..., A_{10}-B_{10}) with 100 messages exchanged within each pair and finally 100 agent pairs (A_1-B_1, A_2-B_2, ..., A_{100}-B_{100}) with 10 messages per pair. Moreover, for each of these configurations two different ways of executing the tests are applied. In the *serial* test, the A agent from each pair sends one message to its B counterpart and when a reply is received, the roundtrip time for this trial is computed. It is repeated in the same manner N-times

(N is 1000/100/10 according to number of agents) and after the N-th roundtrip is finished, the average response time is computed from all the trials. The *parallel* test differs in such a way that the A agent from each pair sends all N messages to B at once and then waits until all N replies from B are received. In both the cases, when all the agent pairs are finished, from their results the total average roundtrip time is computed.

As the agent based systems are distributed in their nature, all the agent platforms provide the possibility to distribute agents on several computers (hosts) as well as run agents on several agent platforms (or parts of the same platform) within one computer. Thus for each platform, three different configurations have been considered: (i) all agents running on one host within one agent platform, (ii) agents running on one host but within two agent platforms (i.e. within two Java Virtual Machines - JVM) and (iii) agents distributed on two hosts. The distribution in last two cases was obviously done by separation of the A-B agent pairs.

The overall benchmark results are presented in the table XVIII. Remind that the results for *serial* tests are in milliseconds [ms] while for *parallel* testing seconds [s] have been used. Different protocols used by agent platforms for the inter-platform communication are also mentioned: Java RMI (Remote Method Invocation) for JADE and FIPA-OS, TCP/IP for ZEUS and $\mathcal{A}$–GLOBE , UDP for JACK, and IIOP for GRASSHOPPER.

To give some technical details, two Pentium II processor based computers running 600 MHz (256 MB memory) with Windows 2000 and Java 2 SDK v1.4.1_01 were used.

Some of the tests, especially in the case of 100 agents, were not successfully completed mainly because of communication errors or errors connected with the creation of agents. These cases (particularly for FIPA-OS and ZEUS platforms) are marked by a special symbol.

More transparent representation of these results in the form of bar charts is depicted in figure 17. The left hand side picture in each row corresponds to the *serial* test with one agent pair while the right hand side picture corresponds to the *serial* test with ten agent pairs. The first row represents tests run on one host, the second row corresponds to test run also on one host but within two JVMs and the third row shows results of testing on two hosts.

7.3. Memory requirements benchmark

Approximate memory requirements per agent can be seen on figure 18. Memory used by JAVA is computed as `java.lang.Runtime.totalMemory()`

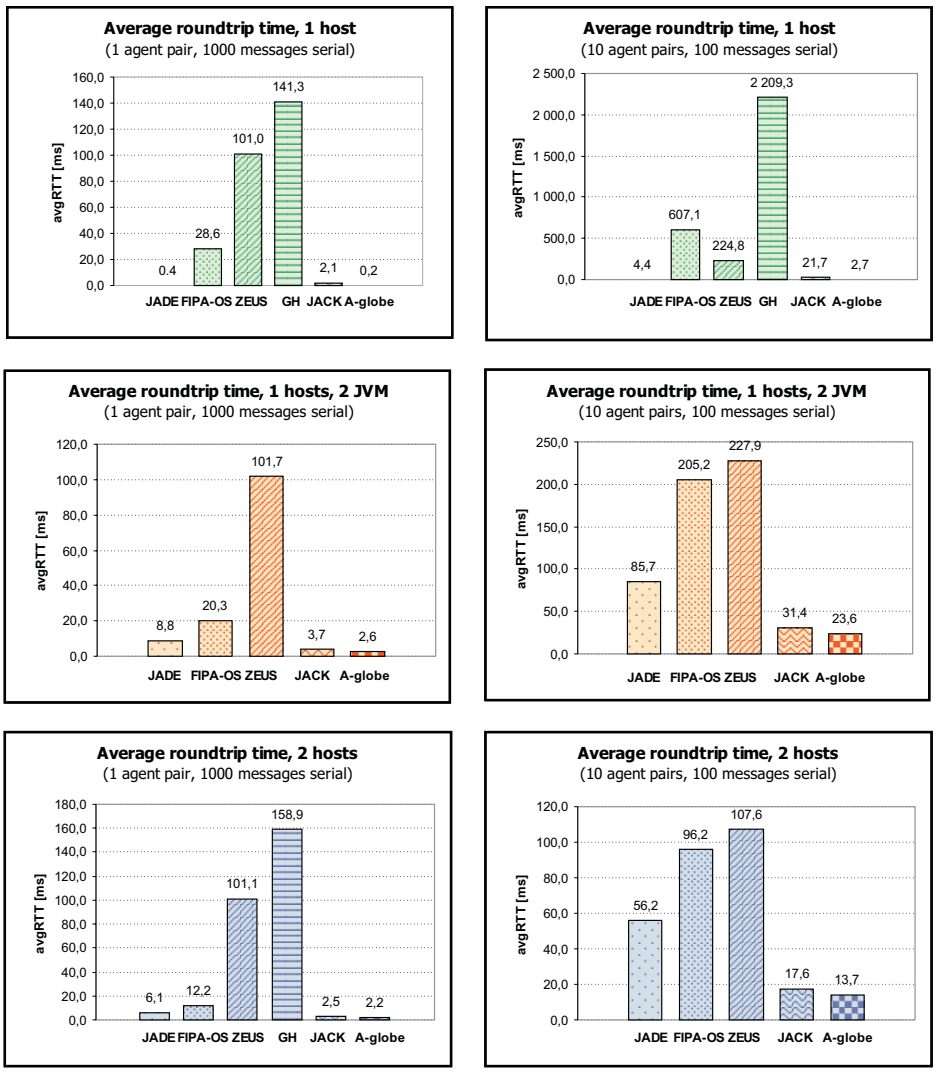

Figure 17. Serial test results: 1^{st} row - one host, 2^{nd} row - one host with two agent platforms, 3^{rd} row - two hosts. Left hand side - 1 agent pair, right hand side - 10 agent pairs

- `java.lang.Runtime.freeMemory()`. Average memory requirement for one agent is obtained as difference between the memory used before and after creation of 100 agents divided by 100.

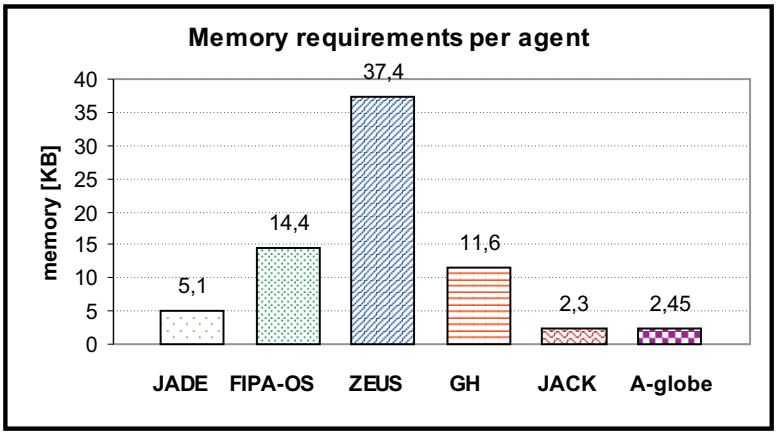

Figure 18. Approximate memory requirements per agent.

7.4. $\mathcal{A}$–GLOBE Summary Statement

As can be seen, $\mathcal{A}$–GLOBE has the best results in all message sending speed benchmarks (table XVIII) from all selected agent platforms. In comparison with its main competitors, JADE and FIPA-OS, the $\mathcal{A}$–GLOBE is at least two times faster than JADE and six times faster than FIPA-OS. $\mathcal{A}$–GLOBE hasn't any communication errors. Also in memory benchmark (figure 18) $\mathcal{A}$–GLOBE has one of the smallest memory requirement per agent.

8. Conclusion

This research project carried out under support of the The "Inaccessibility in Multi-Agent Systems" project (contract no.: FA8655-02-M-4057) has been successful and managed to accomplish the set research targets. Besides the research targets specified in the proposal we have managed to develop a new agent integration platform supporting agent migration and modelling agents inaccessibility.

An important contribution of this research project have been as follows:

- formalization of the problem of agents' inaccessibility,
- classification of various types of agents' inaccessibility,
- suggesting the metrics for measuring agents accessibility and agents inaccessibility,
- design of an alternative mechanism for managing agent social knowledge – the concept of the stand-in agent, and
- development of the $\mathcal{A}$–GLOBE platform for modelling agents' mobility and inaccessibility.

Usefulness of the suggested approach has been illustrated on a series of experiments in the humanitarian aid provisioning scenario. The experiments shown that the stand-in agent contribute have got a highly improving role in the situations with high inaccessibility. The concept of social knowledge maintenance and exploitation has been proven as highly efficient vehicle for handling the inaccessibility problems.

The $\mathcal{A}$–GLOBE platform has been tested and compared with several leading agent integration platforms. The results illustrated the power of the system in terms of its lightweightness and efficiency in message passing. In these aspects the $\mathcal{A}$–GLOBE platform outperformed such platforms like JADE, JACK, etc. This fact has been confirmed by a transparent evaluation carried out by an independent expert.

9. Declaration

The Contractor, Czech Technical University in Prague, hereby declares that, to the best of its knowledge and beliefs, the technical data delivered herewith under the Contract No. FA8655-02-M-4057 is complete, accurate, and complies with all requirements of the contract.

The Contractor certify that there were no subject inventions to declare as defined in FAR 52.227-13, during the performance of this contract.

Table XVIII. Message delivery time results for selected agent platforms

JAVA-based Agent Development Toolkits/Platforms - Benchmark Results January 2004, Rockwell Automation						
PIII, 600MHz, 256MB Agent Platform	**Message sending - average roundtrip time (RTT)**					
	agents: 1 pair messages: 1.000 x ⇆		agents: 10 pairs messages: 100 x ⇆		agents: 100 pairs messages: 10 x ⇆	
	serial [ms]	parallel [s]	serial [ms]	parallel [s]	serial [ms]	parallel [s]
JADE v2.5	0,4	0,36	4,4	0,22	57,8	0,21
JADE v2.5 1 host, 2 JVM, RMI	8,8	4,30	85,7	4,34	1 426,5	4,82
JADE v2.5 2 hosts, RMI	6,1	3,16	56,2	3,60	939,7	3,93
FIPA-OS v2.1.0	28,6	14,30	607,1	30,52	2 533,9	19,50
FIPA-OS v2.1.0 1 host, 2 JVM, RMI	20,3	39,51	205,2	12,50	×	×
FIPA-OS v2.1.0 2 hosts, RMI	12,2	5,14	96,2	5,36	×	×
ZEUS v1.04	101,0	50,67	224,8	13,28	×	×
ZEUS v1.04 1 host, 2 JVM, ?	101,7	51,80	227,9	×	×	×
ZEUS v1.04 2 hosts, TCP/IP	101,1	50,35	107,6	8,75	×	×
GRASSHOPPER v2.2.4b + FIPA addon v1.0	141,3	1,98	2 209,3	0,47	×	×
GRASSHOPPER v2.2.4b 1 host, 2 JVM, ?	N/A	N/A	N/A	N/A	N/A	N/A
GRASSHOPPER v2.2.4b 2 hosts, IIOP	158,9	×	605,7 ×	×	×	×
JACK v3.51	2,1	1,33	21,7	1,60	221,9	1,60 !
JACK v3.51 1 host, 2 JVM, UDP	3,7	2,64	31,4	3,65	185,2	2,24 !
JACK v3.51 2 hosts, UDP	2,5	1,46	17,6	1,28	165,0	1,28 !
AGlobe	0,2	0,07	2,7	0,06	17,7	0,08
AGlobe 1 host, 2 JVM, TCP/IP	2,6	0,26	23,6	0,33	233,8	0,98
AGlobe 2 hosts, TCP/IP	2,2	0,35	13,7	0,39	123,3	0,40

References

1. 'Document Type Definition'. http://www.w3.org/TR/2000/REC-xml-20001006.
2. 'eXtensible Markup Language'. http://www.w3.org/XML/.
3. 'FIPA ACL Message Structure Specification'. http://www.fipa.org/specs/fipa00061/SC00061G.pdf.
4. 'FIPA Communicative Act Library Specification'. http://www.fipa.org/specs/fipa00037/SC00037J.pdf.
5. 'FIPA Interaction Protocol Library Specification'. http://www.fipa.org/specs/fipa00025/XC00025E.pdf.
6. 'IEEE802.11 Specification'. http://grouper.ieee.org/groups/802/11/index.html.
7. 'JAVA API for XML Binding'. http://java.sun.com/xml/jaxb.
8. 'JAVA Forums'. http://forum.java.sun.com/.
9. Cao, W., C.-G. Bian, and G. Hartvigsen: 1997, 'Achieving Efficient Cooperation in a Multi-Agent System: The Twin-Base Modelling'. In: P. Kandzia and M. Klusch (eds.): *Co-operative Information Agents*, No. 1202 in LNAI. Springer-Verlag, Heidelberg.
10. Harder, R., 'Base64 coding and decoding, Java Source'. http://iharder.net/xmlizable.
11. Mařík, V., M. Pěchouček, and O. Štěpánková: 2001, 'Social Knowledge in Multi-Agent Systems'. In: M. Luck et al. (eds.): *Multi-Agent Systems and Applications*, No. 2086 in LNAI. Springer-Verlag, Heidelberg.
12. Pěchouček, M., M., V. Mařík, and J. Bárta: 2002, 'A Knowledge-Based Approach to Coalition Formation'. *IEEE Intelligent Systems* **17**(3), 17–25.
13. Pěchouček, M., V. Mařík, J. Bárta, J. Tožička, O. Štěpánková, and M. Jákob: 2003a, 'Monitoring and Meta-Reasoning in Multi-Agent Systems'. final report to Air Force Research Laboratory AFRL/EORD research contract (FA8655-02-M4056).
14. Pěchouček, M., O. Štěpánková, V. Mařík, and J. Bárta: 2003b, 'Abstract Architecture for Meta-reasoning in Multi-Agent Systems'. In: P. Mařík, Muller (ed.): *Multi-Agent Systems and Applications III*, No. 2691 in LNAI. Springer-Verlag, Heidelberg.
15. Tožička, J., J. Bárta, and M. Pěchouček: 2003, 'Meta-Reasoning for Agents' Private Knowledge Detection'. In: *Klusch, M., Ossowski, S.,Omicini, A., Laamanen, H. (Eds.) Cooperative Information Agent VII – Lecture Notes in Computer Science, LNAI 2782*. Heidelberg : Springer-Verlag.
16. Vrba, P.: 2003, 'JAVA-Based Agent Platform Evaluation'. In: V. Mařík, D. McFarlane, and P. Valckenaers (eds.): *Holonic and Multi-Agent Systems for Manufacturing*, No. 2744 in LNAI. Springer-Verlag, Heidelberg.
17. Walrath, K., M. Campione, and A. Huml: 2003, *The JAVA Tutorial*. Sun Microsystems, Inc. http://java.sun.com/docs/books/tutorial/index.html.